Lecture 1: Introduction
This Course

• Programming Language Technologies and Paradigms
  – More colloquially: advanced programming

• This semester (as for the last several years): concurrency and parallelism
Context

• Chips aren’t getting faster, they are getting more computing cores

• Gain in compute power by running on many machines, e.g., “the cloud”

• Thus: would like to write programs that are concurrent, parallel, and distributed: more complicated than sequential programs
Course Goal

• How to build **correct** and **efficient concurrent and parallel** programs
  – Focus on the **Java** programming language
    • Classic support for **shared-memory concurrency**
    • Concepts should be more general
  – We will also consider **Erlang**
    • Functional language, gaining popularity
    • Uses **message-passing concurrency**, by and large
Textbook: JCIP

• Most of the semester will use this book
• Will rely on this book for written explanations of concepts; will provide examples on-line
• Lecture notes for Erlang and other topics
Coursework and assessment

• Five projects, 10% each
• Two midterms, 12% each
• One final exam, 25%
• Meet your professor, 1%
  – Come to my office for five minutes to chat about the course, your career goals, your favorite programming language, …
Projects

• Learn by doing!
  – Thus, you need to do them yourself to really learn
• Must submit by midnight on due date
  – May submit up to 24 hours late for a 20% penalty
• Grading comes from tests and manual inspection
  – You will often write a document to explain your design and why it works
Bonus prizes

• At least one project will have an optional competition
  – The winner(s) will get a cool (geek) T-shirt (of their choice)
• E.g., third project: most effective word finder
  – We’ll run it on a 24 processor machine to judge
  – You can test on Linuxlab multi-proc machines
Projects Computing Environment

• Java 7
  – JUnit 4+ used for testing
• Eclipse 4.3 (Kepler)
  – Installation help: www.cs.umd.edu/eclipse
• CS submit server
• Linuxlab accounts
  – Make sure your project works here first

Later on in course: Erlang
Exams

• Midterm #1 – October 14
• Midterm #2 – November 20
• Final (comprehensive) – December 19

• Make a note of these dates now!
Communication

• **Announcements** and **Q&A using Piazza**
  – Link from the course site

• **E-mail** should be **reserved for special cases**
  – Excused absences due to illness, etc.
  – Project questions should go to Piazza, so all can see the answers
Remaining items of syllabus

• See www.cs.umd.edu/class/fall2013/cm sc433 for policies on
  – Excused absences
  – Accommodating students with disabilities
  – Policies on academic integrity (cheating)
    • In short: you may discuss ideas, but not share code
    • But beware: if you let someone else spoon-feed you the “trick” to the project, you will be less prepared for the exams
Schedule

• Remainder of today: JCIP Chapter 1
• Next time: JCIP Chapter 2 (read chapter in advance of class)
Concurrency

• **Sequential** execution: At any point during execution, at most one instruction can be executed next

• **Concurrent** execution: Several possible instructions can be executed “next”

• **Processes** and **threads** are common abstractions for concurrent execution
  – Roughly: Processes do not share resources (in particular, memory), threads do
Concurrency vs. Parallelism

• Parallelism is a special form of concurrency
  – **Concurrency** is when two tasks can start, run, and complete in overlapping time periods.
    • They need **not** ever run **at the same instant**, e.g., concurrency by **time-slicing** on a uni-processor
  – **Parallelism** is when tasks literally **run at the same time**, e.g. on a multicore processor
    • Hence it’s a specific kind of concurrency
    • You need parallelism for **asymptotic** speedups
Why Concurrency?

• Resource utilization/performance
  If they can do operations simultaneously, applications run faster! E.g., I/O and CPU busy at the same time

• Convenience
  Sometimes easier to specify a computation as cooperating/concurrent parts, than to merge things all together (e.g., a GUI, “blocking” I/O)
Threads

• Threads are **independent activities** that **share resources**, notably memory
  – Hence threads implement **shared-memory concurrency**

• Operationally
  – Each thread has its own **register set** (including the **program counter**) and a **stack**
  – All threads **share** the same **heap**
Benefits of threads

• **Performance**
  – True *parallelism* (e.g., parallel mergesort)
  – **Asynchrony** (e.g., one thread blocks on I/O while another can move ahead)
  – **Responsiveness** (separate GUI thread)

• **Simplicity** of modeling
  – One thread per task
    • Imagine writing a single-threaded vs. a multi-threaded simulation
Hazards of threads

- **Safety** hazards due to *nondeterminism*
  - Race condition
  - Atomicity violation (often caused by a race)
  - Can fix safety hazards using *synchronization*

- **Liveness** hazards due to *over-synchronization*
  - Deadlock

- **Performance** hazards
  - Not as bad as deadlock, but still too slow
Why Is (Threaded) Concurrency Hard?

• Nondeterminism!
  – Executing same program can yield different answers
  – Replaying a given execution is very difficult

• Concurrency breaks *procedural abstraction*
  – Procedural abstraction: a given sequence of instructions will always return the same result if started in the same state
  – Implication: you can think of a sequence of instructions as conceptually a single instruction
  – Basis for: compilation, method definition, etc.
Nondeterminism

• Suppose we have
  – Shared variable `shared` that is initially 0
  – Two threads `t1` and `t2` with instance variables `myShared`, each of which does:
    ```
    myShared = shared;
    myShared++;
    shared = myShared;
    ```

• What are possible values of `shared` afterwards?
  – 1, 2!
Atomicity

• Consider previous example, and suppose threads were launched via following:
  
  ```java
  t1.start();
  t2.start();
  ```

• If each thread’s execution is atomic, then
  
  – `t1.start()` is conceptually a single operation that increments shared
  – So is `t2.start()`
  – Only allowed answer would be 2!
Synchronization

• You can use synchronization to enforce atomicity
• Doing so is safe, but may lead to liveness and/or performance hazards
• Will cover more next time

• See BadCounter.java example
Next time: JCIP Ch. 2

• Thread safety
• Atomicity
• Locking